# A Network Architectural Style for Real-time Systems: NaSr

R. Bashroush, I. Spence, P. Kilpatrick, T.J. Brown
*Queens University Belfast*
*School of Computer Science*
*18 Malone Road, Belfast BT7 1NN, UK*
*{r.bashroush, i.spence, p.kilpatrick, tj.brown}@qub.ac.uk*

## Abstract

*Inter-component communication has always been of great importance in the design of software architectures and connectors have been considered as first-class entities in many approaches [1][2][3]. We present a novel architectural style that is derived from the well-established domain of computer networks. The style adopts the inter-component communication protocol in a novel way that allows large scale software reuse. It mainly targets real-time, distributed, concurrent, and heterogeneous systems.*

## 1. Introduction

Our experience with ADLARS[1] [4] in designing the software architecture of small to medium case studies [5] showed that component (or *Task* as it is called in ADLARS) communication can get very complicated, especially in the case of concurrently executing components with both synchronous and asynchronous communication interfaces within a heterogeneous environment. These case studies highlighted some issues in the context of component communications. In addition, a major part of the architecture evaluation stage was found to involve analysis of inter-component dependences and consistencies.

During the course of our work we noticed a similarity between our domain of concern and the computer networking domain. Networks run different nodes (computers) executing concurrently using synchronous and asynchronous communication within a heterogeneous environment (nodes running under different OSs for instance). Inspired by peer-to-peer architectural styles, the idea led us to the development of an architectural style called NaSr that adopts the internetworking discipline with some modifications that emphasize few software architecture concepts that do not exist in the networking domain [6].

The next section presents the overall framework of the NaSr architectural style. Related work is treated in section 3. Section 4 summarizes the current foreseen limitations and challenges within the style. Conclusion and future work are finally shown in section 5.

## 2. Network Architectural Style for Real-time systems, NaSr

In this section we present an overview of the NaSr style and its framework.

The NaSr Framework consists of:

- Components
- Connection handlers
- Communication Protocols

NaSr architectures consist of concurrently executing OTS or user defined components wrapped inside NaSr *Components* (Section 2.1) that utilize a packet driven method of communication using defined *communication protocols* (Section 2.3). The communication management is looked after by *Connection handlers* (Section 2.2).

In the following, we use the term Component to refer to NaSr Components (OTS or user defined Component(s) plus a *Domain Adapter*, Figure 1).

Within NaSr, every component is identified by a *unique ID* and provides/requires a specific set of *service(s)*. This is a key feature of the NaSr style that allows the separation of the services provided/required in the system from the components providing them. The separation allows any component in the system to be replaced (due to failure) or backed up (due to overload) by another component(s) that provides the

[1] Architecture Description Language for Real-time Systems: an ADL that was developed within our research group first in 1999 as part of the Jigsaw project funded by Nortel Networks ®.

same set of services without the need to reconfigure or restart the system. The newly added component(s) can make itself known to the Service Translation Center STC (Section 2.2.1) by sending an appropriate *registration* message identifying the services it requires/provides. Then new calls for that given service will be routed by the *Connection handlers* to the newly added component. The reader can see here the solutions and scenarios adopted from the real networking domain. This architecture strongly supports system's reconfigurability and increases system uptime. Also notice the separation of connection management from computational components.

Components are described in 2.1, connection handlers in 2.2, and communication protocols in 2.3.

## 2.1. Components

A component in NaSr is a separate thread of execution. Each component wraps a user defined or OTS component (Figure 1) that can be developed using any language and employ any interface types (event based, message based, etc.). The communication can be utilized by employing a NaSr *Domain Adapter* that translates the wrapped component's interface to NaSr packet based communication following a desired protocol.

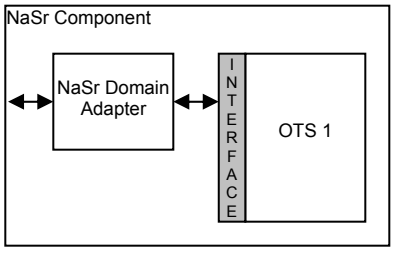


**Figure 1.** A NaSr Component wrapping an OTS Component (OTS 1) communicating with the system via the NaSr Domain Adapter

## 2.2. Connection handlers

Connection handlers are the objects of the NaSr style that handle the packet driven communication among components. Currently, we have identified three communication handling objects:

- Service Translation Center
- Communication Manager
- Broadcaster

These objects form the backbone for component communication using user-defined protocols. More objects and protocols can be developed in the future to allow more complicated communication services. The three objects are discussed next.

**2.2.1. Service Translation Center (STC).** This is a key object for building architectures using the NaSr style. It provides a translation table between services and the components providing them.

In general, the role of the STC in NaSr architectures is similar to the role of a DNS (Domain Name System) on an IP network. DNS provides translations between domain names and IP addresses whereas an STC translates between service names and components providing them.

The STC proved to be very useful in consistency and dependency architecture analysis for architecture verification.

**2.2.2. Connection Manager (CM).** The Connection Manager plays a similar role to a regular network router which forwards packets back and forth among the different machines on a given network.

In NaSr, the CM achieves this with the help of the STC. In a typical scenario, a component asks the CM for a desired service, the CM queries the STC and gets an ID for a component that provides this service. This ID is then passed to the requesting component and a direct communication among the two components takes place.

**2.2.3. Broadcaster.** A Broadcaster is similar to an Ethernet HUB. It can work without the need for an STC (compared to a Connection Manager) by broadcasting incoming packets to all components on the network. This could be useful on smaller architectures that do not require the overhead of an STC and packet routing. When a broadcaster is deployed to connect a group of components, the component communication will be similar to the Event-based Integration style [7].

## 2.3. Communication Protocols

Different communication protocols can be designed to serve different domain functional and nonfunctional attributes. For example, in concurrent time-critical systems, a time-stamp field would be included within the packet header to enable communication synchronization. Our aim is to have a library of protocols designed for different application domains that an architect can choose from.

In one system, we can have more than one protocol in use at the same time. To better understand that, consider two different sub-nets running different protocols to best suit their applications, but still they can communicate via a backbone structure.

At the moment, we have designed one communication protocol (the P256 Protocol) that is inherited from the well-developed TCP/IP protocol stack but tailored to suit the Software Architecture domain.

Our packet consists of a header and a payload. The payload carries the message sent from one component to another, and the header contains the information necessary for routing the message to its desired destination. The different fields of the header and their descriptions are shown in Table 1.

| Field Name | Description |
|---|---|
| *Protocol Identifier (3 bits)* | could be used for compatibility issues when a new protocol is developed, or more than one protocol is deployed in the system |
| *Message type (3 bits)* | specifies whether the message contained is a:<br>1. Registration<br>2. Unknown Service Provider<br>3. Service Providers Request<br>4. Direct Communication<br>5. Refresh<br>6. Overload<br>7. Error |
| *Target Service Name (152 bits)* | the service name required by the source component |
| *Source ID (24 bits)* | the *Component_ID* of the source component |
| *Destination ID (24 bits)* | the *Component_ID* of the receiving component (usually filled at the CM) |
| *Time Stamp (48 bits)* | In time-critical systems, this field is used to monitor routing delays. |
| *Importance (2 bits)* | could be a number between 1-4 for showing the importance of this packet as a possible future option for implementing Quality of Service in interactions (QoS) |

**Table 1.** The NaSr P256 Protocol Packet header fields

## 3. Related work

The NaSr style draws its main idea, the packet-based inter-component communication, from the computer networking domain. That is due to the similarity between NaSr's domain of interest (real-time, heterogeneous and concurrent) and the domain of computer networking.

Also, it shares a lot of concepts with Brokered Distributed Object systems [8] similar to CORBA within the OMG [9] and Open Distributed Processing (ODP) within ISO/IEC [10] which utilize name resolver components (similar to the concept of *Connection Handlers* in NaSr).

Being a peer-to-peer style, the work on NaSr gained from the experience (both good and bad) of many other researchers working with similar styles. Two of these peer-to-peer styles are discussed next.

### 3.1. Event-Based Integration EBI (or implicit invocation)

Within the EBI style [7], components do not invoke other components directly; instead, component communication is attained by event broadcasting. Other components with interest of this type of events can register their interest and then be executed by the system itself when the specific event type is fired. This method of communication reduces component coupling leading to better support for extensibility, reuse and evolution [7].

However, this style of communication raises many other issues like scalability, event storms, single point of failure, and lack of event response anticipation [7].

### 3.2. C2

The C2 architectural style [11][12], designed originally targeting GUI applications, combines the EBI style with the layered-client-server style [7][13] to support large-grain reuse and flexible composition of system components by enforcing substrate independence. Components communicate using asynchronous message passing up (requests) and down (notifications) the layered system to enforce loose coupling of components at higher levels, and uncoupling at lower layers.

With this layered structure, no component can broadcast a message to all the other components within the system (no requests can be sent down the hierarchy). This can be a critical drawback in some application domains (even though it might not be the case with GUI applications).

## 4. Current limitations and challenges

Some case studies have been designed and constructed using the NaSr style; however, it is still under continuous development with open issues and more research questions to be asked. This is the case with any newly developed architectural style where a considerable amount of time is required for adequate analysis of all the style aspects and identification of the required improvements.

- The case studies we developed were mostly small-scale systems. The style is intended to perform best with large-scale systems where the overhead introduced by the communication management and protocols is acceptable. This overhead could be questionable with small-scale systems.
- Deciding upon whether *Connection Handlers* are architectural elements and should be shown at the architecture level of the system or not remains an open question.
- Fine tuning the level of abstraction. Shall the STC carry information about a component such as its placement (in multi-processor environments) and memory location, or is that considered irrelevant at our level of abstraction?

## 5. Conclusion

With the increasing level of complexity of newly emerging real-time, concurrent, and heterogeneous systems, and with the flourishing OTS marketplace, the need for a well constrained communication framework that facilitates OTS integration becomes highly desirable.

Also, with today's system-on-chip implementations, component communication can get very complicated to implement, and a packet-driven inter-component communication proves to be a potential solution.

Until now, small to medium scale case studies were implemented to asses and fine tune the different aspects of the style. The outcomes of these exercises were very encouraging.

The NaSr style is under continuous development, and in addition to experimenting with style and enhancing its specification, along with the design of more case studies, the construction of a NaSr development environment is on our future task list.

## 6. Acknowledgments

This research is based on projects funded by British Telecom and Nortel Networks.

We would like to thank the WICSA reviewers for their valuable and useful comments.

## 7. References

[1] R. Allen and D. Garlan. Formalizing Architectural Connection. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 71-80, Sorrento, Italy, May 1994.

[2] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstraction for Software Architecture and Tools to support Them. *IEEE Transactions on Software Engineering*, volume 21, number 4, pages 314-335, April 1995.

[3] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, Vol 1, No 4, pp. 355-398, October 1992

[4] T.J. Brown, I. Spence, and P. Kilpatrick. ADLARS: A Relational Architecture Description Language for Software Families. *Proceedings of the Fifth International Workshop on Product Family Engineering*, Siena, Italy, 2003.

[5] R. Bashroush, I. Spence, P. Kilpatrick, and T.J. Brown. A Real-time Network Emulator: ADLARS Case Study. *Proceedings of the Third Asia Pacific International Symposium on Information Technology*, pages 610-617, Istanbul, Turkey, Jan 2004.

[6] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, Vol 17, No 4, pages 40-52, October 1992.

[7] D. Garlan and M. Shaw. An introduction to software architecture. Ambriola & Tortola (eds.), *Advances in Software Engineering & Knowledge Engineering*, vol. II, World Scientific Pub Co., Singapore, 1993, pp. 1-39.

[8] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture: A system of patterns*. John Wiley & Sons Ltd., England, 1996.

[9] Object Management Group. *The Common Object Request Broker: Architecture and Specification (CORBA 2.1)*. <http://www.omg.org/>, Aug. 1997.

[10] ISO/IEC JTC1/SC21/WG7. *Reference Model of Open Distributed Processing*. ITU-T X.901: ISO/IEC 10746-1, 07 June 1995.

[11] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., and J. E. Robbins. A Component- and Message-Based Architectural Style for GUI Software. In proceedings of *the 17th International Conference on Software Engineering*, pages 295-304, Seattle, WA, April 1995.

[12] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., and J. E. Robbins. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, Vol 22, No 6, pages 390-406, June 1996.

[13] A. Sinha. Client-server computing. *Communications of the ACM*, 35(7), July 1992, pp. 77-98.